Master Thesis

Advection of sampling grids for efficient computation of trajectory-based quantities

Author(s):
Rigazzi, Alessandro

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005575240

Rights / License:
In Copyright - Non-Commercial Use Permitted
Advection of Sampling Grids for Efficient Computation of Trajectory-Based Quantities

Alessandro Rigazzi

Master Thesis
April 2008

Prof. Dr. Markus Gross
Abstract

Lagrangian methods in numerical flow visualization, such as the finite-time Lyapunov exponent, often require the integration of a very high number of trajectories. Usually one trajectory needs to be computed for each node of the sampling grid. This constitutes a very high cost especially for transient vector fields because there the trajectories need to get recomputed for each time step. This thesis addresses the problem in two ways. As a first improvement to the known methods we restrict sampling grids to areas of interest, where phenomena which are to be visualized appear, disregarding the rest of the domain. The second improvement is done by exploiting temporal coherence of trajectory-based quantities by reusing part of the trajectories. This is achieved by advecting the nodes of the sampling grid. As a case study, height ridges of finite-time Lyapunov exponent are extracted on synthetic Computational Fluid Dynamics datasets provided by partners of the ETHZ. Performances of the method are tested against those of a classic method for computation of such ridges.
Acknowledgments

I would like to thank my family, for supporting me during all these years of studies, professors Markus Gross and Ronald Peikert for giving me the opportunity of researching in a field I always feel excited about, Filip Sadlo for his continuous help throughout all this work, and Gioacchino Noris for his invaluable patience with illustrations.
1. Introduction 1
   1.1. Vector Field Topology 1
       1.1.1. Vector Fields, Ordinary Differential Equations and Flows 1
       1.1.2. Classification of Trajectories, Stationary Points and Manifolds 3
   1.2. Lagrangian Coherent Structures 5
       1.2.1. Finite-Time Hyperbolic Material Surfaces and Lines 6
       1.2.2. The Analytic Approach 6
   1.3. Finite-Time Lyapunov Exponent 7
   1.4. Height Ridges and Valley Lines 10
       1.4.1. Conditions for Ridge Existence 10
       1.4.2. Ridge Extraction and Filtering 10

2. Related Work 13
   2.1. Height Ridges 13
   2.2. Adaptive Mesh Refinement 14
   2.3. Measures of Mesh Deformation 14
   2.4. Importance-Driven Particle Techniques 14
   2.5. Newtonian Coherent Criteria 14
   2.6. Objective Definition of Vortices 15

3. Method 17
   3.1. Motivation 17
# List of Figures

1.1. Stationary Points ................................................. 4  
1.2. Hyperbolic Material Lines and Surfaces ............................. 6  
1.3. Flow Separation and Flow Attachment ............................. 9  

3.1. Reference Grid .................................................. 20  
3.2. Grid Resampling .................................................. 21  
3.3. Growing the Grid .................................................. 24  
3.4. Growing Toward Walls ............................................ 25  
3.5. Subdivision ...................................................... 27  
3.6. Pruning the Grid .................................................. 28  
3.7. Grid Advection .................................................... 29  
3.8. Complete Algorithm .............................................. 31  

4.1. Von Kármán Vortex Street, Geometry and Grid ..................... 34  
4.2. Von Kármán Vortex Street, Ridge and Trajectories ................ 35  
4.3. Flow around a Cuboid, Geometry and $\lambda_2$ ...................... 36  
4.4. Flow Around a Cuboid, Grid and FTLE .......................... 37  
4.5. Intake of a Water Turbine, Geometry and Uniform Grid ........... 38  
4.6. Intake of a Water Turbine, Grid and Ridge ...................... 39  
4.7. Computation Time ............................................... 39  
4.8. Number of Nodes and Cells ..................................... 40  
4.9. Volume Distortion for Cuboid Data Set .......................... 41  
4.10. Volume Distortion for Intake Data Set .......................... 42  

5.1. Trajectories Storage .............................................. 46  

A.1. AVS Module ...................................................... 51
List of Figures

A.2. AVS Control Panel ........................................... 54
List of Tables

4.1. Performance Analysis for Kármán Dataset ........................................ 34
4.2. Performance Analysis for Cuboid Dataset ......................................... 36
4.3. Performance Analysis for Turbine Intake Dataset ............................... 37
List of Tables
Introduction

This thesis aims at improving the state of the art methods for computation of trajectory-based quantities. In particular, it is applied on time-dependent vector fields for ridge extraction from finite-time Lyapunov exponent. In this chapter we introduce the mathematical setting of the problem, the concepts that will be relevant for the rest of the thesis, and the terminology that we will use.

1.1. Vector Field Topology

Vector Field Topology (VFT) was introduced to the visualization community by Helman and Hesselink in [HH89]. An intuitive definition of VFT is given in [Asi93], where it is depicted as the description of important topological features and concepts that can be used to get a glimpse of “what a vector field is doing”. In Section 1.1.2 we illustrate some of these features, but first, in Section 1.1.1, we introduce definitions for vector fields and their principal constituents. The source for this section is [Asi93], the interested reader is referred to the original document for a deeper description and a more precise mathematical analysis of the exposed subject.

1.1.1. Vector Fields, Ordinary Differential Equations and Flows

For our work we are interested in three definitions, those of a vector field, an ordinary differential equation, and a flow. Each definition builds on top of the previous one and will be useful for the rest of the thesis.
1. Introduction

Vector Field

If we assign a vector to each point of a given space, then we obtain what is called a vector field. In general, let $U$ denote an open set of Euclidean space $\mathbb{R}^n$. Then a vector field on $U$ is a function of the type $u : U \to \mathbb{R}^n$. One assumption we have to make is that the vector field $u$ is at least once continuously differentiable, denoted $C^1$. If the vector field varies in time, then it is called a time-dependent vector field, and it is defined as $u : U \times I \to \mathbb{R}^n$, where $I$ is some time interval.

Ordinary Differential Equation

An autonomous, ordinary differential equation (ODE) defined from $U$ to $\mathbb{R}^n$ is an equation of the form

$$\frac{dx}{dt} = u(x)$$

where $x : I \to U$ denotes an unknown curve parametrized by some interval $I$ containing 0, and $u : U \to \mathbb{R}^n$ is $C^1$.

Given any point $u(x_0) \in U$, we assume an initial condition of the form $x(0) = x_0$. This condition allows us to expose the Existence Theorem for solutions of autonomous ODE’s:

*There exists some number $c > 0$ and a solution $x : (-c, c) \to U$.*

In other words, $x(0) = x_0$ and, for each $t$ in the interval $(-c, c)$, we have $x' = u(x(t))$. This solution is unique, i.e., every curve $y$ satisfying the initial condition $y(0) = x_0$ and $y' = u(y(t))$ will be exactly the same curve as $x$. This is called the Uniqueness Theorem for solutions of autonomous ODE’s.

One can think of the time-dependent versions of the ODE’s, these are called non-autonomous differential equations and they are equations of the form

$$\frac{dx}{dt} = u(x, t)$$

where $u : U \times I \to \mathbb{R}^n$ is a $C^1$ function taking two arguments. The solution of such an ODE is similar to that for autonomous ODE’s, where the time interval becomes $(t_0 - c, t_0 + c)$.

Flow

Each trajectory which takes the value $p$ at time $= t_0$ must satisfy the fundamental consistency condition:

*let $p \in U$ and let $s$ and $t$ denote any two lengths of time. Then following the trajectory of $p$ for time $s$, and following the trajectory of the reached point for time $t$ starting at time $t_0 + s$ or following the trajectory of $p$ for time $s + t$ give the same result.*

For any $p \in U$ and $[t_0, t_0 + t] \in I_p$, let us define $\phi^{t_0+t}_t(p)$ as the point reached following, for a time $t$, the trajectory starting in $p$ at time $= t_0$ ($I_p$ is the interval of time where this trajectory is
1.1. Vector Field Topology

defined). We can now write the consistency condition as

\[ \phi^{t_0+s+t}(\phi^{t_0+s}(p)) = \phi^{t_0+s+t}(p). \]  (1.3)

It also trivially holds \( \phi^{t_0}_0(p) = p \). The existence and uniqueness theorem for ODE’s impose that the flow function \( \phi \) of the variables \( p, t_0 \), and \( t \) is \( C^1 \).

For each fixed value of \( t \) and starting time \( t_0 \), there is a mapping of \( U \) onto itself which takes \( p \) to \( \phi^{t_0+t}_0(p) \). We will denote this mapping \( \phi^{t_0+t}_0 \) and refer to it as to the flow map. This concept will be useful in its discretized version for the definition of the finite-time Lyapunov exponent in Section 1.3.

1.1.2. Classification of Trajectories, Stationary Points and Manifolds

In this section we see the principal topological features of vector fields, they will be of interest for the study of the phenomena this thesis addresses. We will restrict this section to 3-dimensional spaces, since these are the spaces we will compute on.

Classification of Trajectories

As a consequence of the Uniqueness Theorem, in a \( C^1 \) vector field, at a given time \( t \), each point \( p \in U \) lies on one and only one trajectory. Moreover, trajectories can never converge, cross each other or branch.

We can therefore classify trajectories into three types:

- **regular trajectories**, when the curve never returns to a position where it has already been;
- **closed orbits**, when the mapping has a period > 0, that is, if the period is denoted by \( t_0 \), it holds \( x(t + t_0) = x(t) \);
- **stationary points**, when the entire trajectory is a single point.

Classification of Stationary Points

If the point \( p \) is a stationary point of a 3-dimensional vector field

\[ u(x, y, z) = \left( u(x, y, z), v(x, y, z), w(x, y, z) \right) \]  (1.4)

then we can examine the Jacobian matrix of partial derivatives \( J = (\partial u_i/\partial x_j) \) in order to examine the trajectories of neighboring points. We can then calculate eigenvalues and eigenvectors belonging to \( J_p \) (the Jacobian matrix evaluated at the stationary point \( p \)).

We define a stationary point \( p \) as **hyperbolic** if the real parts of the eigenvalues of \( J_p \) are all non-zero. Hyperbolic stationary points are important, because they exhibit local structural stability, i.e., a local perturbation of a hyperbolic stationary point does not change the nearby topology.
1. Introduction

![Image: Four types of stationary points: from left to right, source, spiral source, 2:1 saddle, and 2:1 spiral saddle. To obtain the remaining four types, it suffices to reverse the arrows (image taken from slides for the Scientific Visualization lecture given at ETHZ by Professor Ronald Peikert).]

We classify stationary points, from a topological point of view, by the number of positive and the number of negative real parts that the eigenvalues have. Recall that we are computing eigenvalues of a $3 \times 3$ matrix. These are defined as the roots of the characteristic polynomial $P(J)$ defined by $P(J) = \det(J - \lambda I)$. Since this is a cubic polynomial with real coefficients, there are either 3 real roots or 1 real root and a pair of complex conjugate roots.

The possible cases are:

1. All 3 roots real:
   a. All positive: source
   b. 2 positive, 1 negative: 1:2 saddle (1 dim. in, 2 out)
   c. 1 positive, 2 negative: 2:1 saddle (2 dims. in, 1 out)
   d. All negative: sink

2. 1 real, 1 complex conjugate pair (where the conjugate pair is $a + Li$ and $a - Li$ with $a$ and $b$ real):
   I. Real root positive:
      a. $a$ positive: spiral source
      b. $a$ negative: 2:1 spiral saddle (2 dims. in, 1 out)
   II. Real root negative
      a. $a$ positive: 1:2 spiral saddle (1 dim. in, 2 out)
      b. $a$ negative: spiral sink

Figure 1.1 shows four types of stationary points, the other four are obtained by reversing the arrows.
1.2. Lagrangian Coherent Structures

Manifolds of Hyperbolic Stationary Points

Stable manifolds are topological features of hyperbolic critical elements (which can be stationary points or closed orbits, but we don’t include closed orbits in this tractation, because they are less relevant for our work). As stated in [HH89], these manifolds are also called separatrices, because they separate regions of different flow behavior in the respective direction of time. Therefore their detection and classification are of high importance for the description of steady (time-independent) vector fields.

In topology, with n-manifold, it is meant a space each point of which has a neighborhood that is topologically equivalent to an open neighborhood in Euclidean space \( \mathbb{R}^n \). In our case (the 3-dimensional case), a source \( p \) has a 0-dimensional stable manifold. If \( p \) is a hyperbolic stationary point of a vector field \( u \), the stable manifold of \( p \), denoted by \( W^s(p) \) is the set of points whose trajectories approach \( p \) as \( t \to \infty \). Conversely the unstable manifold of the point \( p \), denoted as \( W^u(p) \), is defined as the stable manifold of \( p \) for \( t \to -\infty \). These are computed integrating streamlines or stream surfaces starting in the direction of the eigenvectors of the velocity gradient at \( p \). One important quality of stable and unstable manifolds is that they are invariant under the action of the flow, i.e., for a point \( p \) which belongs to such a manifold at a certain time \( t_0 \), in a steady vector field, the mapped point \( \phi^{t_0+t}_t(p) \) will belong to the manifold at time \( t_0 + t \).

A 1:2 saddle has a 1-dimensional stable manifold (the point \( p \) and two trajectories along one eigendirection). A 2:1 saddle has a 2-dimensional stable manifold (the point \( p \) and the trajectories which get arbitrarily close to the negative real part eigenplane as \( t \to \infty \)). Finally, a sink \( p \) has a 3-dimensional stable manifold.

1.2. Lagrangian Coherent Structures

As mentioned in [SP07b], one important drawback of stable and unstable manifolds detection is that this is meaningful only for steady vector fields. One reason for this limitation is that streamlines (curves starting at a given point, obtained integrating a steady vector field, or a snapshot of an unsteady vector field taken at a certain time) usually diverge from pathlines (curves described by the true path a massless particle follows). Moreover, critical points often move in unsteady vector fields. Therefore time-dependent vector fields are often analyzed by application of VFT to isolated time steps, but this is often counterintuitive and gives no clue about the true behavior of the vector field or dynamic system. For these reasons other approaches have to be found, in order to describe time-dependent vector fields.

In [Hal01], George Haller shows the utility of Lagrangian Coherent Structures (LCS’s) for vector field analysis and some of the mathematical and physical features they exhibit. We summarize in this section some of these features, this will explain why many people are in the progress of choosing LCS’s as the tool for visualization of behavior of unsteady vector fields. For the remaining features and a deep mathematical analysis of LCS’s we refer the interested reader to the original paper by Haller.
1. Introduction

Figure 1.2.: Hyperbolic material lines and surfaces: different types of finite-time hyperbolic material lines and surfaces containing a trajectory $x(t)$. The figures show instantaneous positions of these lines and surfaces at time $t$ with $x(t)$ appearing as a single point (image taken from [Hal01]).

1.2.1. Finite-Time Hyperbolic Material Surfaces and Lines

We consider again (1.2), with $t \in I$ which is a finite-time interval and $u$ is $C^1$.

We define a material line $L(t)$ as a smooth curve of fluid particles advected by the velocity field and a material surface $M(t)$ as a smooth surface of particles, again advected by the velocity field. Let $I$ be an open time interval within $I$. We call a material surface attracting over $I$, if it is robust for times taken from $I$, i.e. small perturbations from $L(t)$ and $M(t)$ should result in exponential convergence to these material objects while $t \in I$. Similarly, a material object (surface or line) repelling over $I$, is an object attracting over $I$ in the reversed flow. Finally we group the described attracting and repelling material lines or surfaces and say they are finite-time hyperbolic over $I$. Notice that if the length of $I$ tends to infinity, attracting material lines and surfaces become unstable manifolds. In Figure 1.2, the possible typologies of hyperbolic material lines and surfaces are shown.

1.2.2. The Analytic Approach

A Lagrangian coherent structures, denoted by $C(t)$, is distinguished in one of the following two ways:

1. $C(t)$ retains its stability type (attraction or repulsion) for locally the longest time in the flow.
2. $C(t)$ retains its stability type for locally the shortest time in the flow.

Case 1 includes all generic coherent structures in an open flow. Case 2 is relevant for flows with a no-slip boundary, which is the case for the CFD simulations we used as input for our algorithm. In this case some coherent structures, such as those near separation or reattachment points, may be attached to the wall. Parts of these structures which are close to the wall can only be finite-time hyperbolic for short times, since, if they attach to the wall, they can not be finite-time hyperbolic for any length of time (the linearized flow close to a no-slip boundary is degenerate). $C(t)$ will nevertheless exhibit finite-time hyperbolicity for locally the shortest time in a vicinity of the wall, since trajectories starting nearby will pass by the wall and accumulate hyperbolicity time in their later history.

Given an initial condition $x_0$, we can define the scalar functions $T_{LN}(x_0; t_0, t)$, $N = 1, \ldots, 4$, as the total length of time within $[t_0, t]$ over which the trajectory starting at time $t_0$ satisfying the initial condition $x_0$ and denoted by $x(t; t_0, x_0)$ lies in a material line of given stability type LN (cf. Figure 1.2). Similarly, the scalar functions $T_{SM}(x_0; t_0, t)$, $M = 1, \ldots, 6$, gives the total length of time within $[t_0, t]$ over which a trajectory $x(t; t_0, x_0)$ lies in a material surface of stability type SM. A Lagrangian coherent structures can then be defined by the following statement:

*coherent structures are characterized by local extrema in the scalar fields* $T_{LN}(x_0; t_0, t)$ *and/or* $T_{SM}(x_0; t_0, t)$ *for some* $M$ *and* $N$.

The problem behind this approach is that the two scalar fields $T_{LN}(x_0; t_0, t)$ and $T_{SM}(x_0; t_0, t)$ should be computed by verifying the stability of all possible material surfaces and lines in the flow, but this is unfeasible. Therefore, recently, recurring to an actually popular quantity is proposed. This quantity is the finite-time Lyapunov exponent, which allows to compute these scalar fields using only invariants on the velocity gradient along the trajectory $x(t; t_0, x_0)$. We explain the use of this quantity in the next section.

### 1.3. Finite-Time Lyapunov Exponent

The approach we chose for our method was presented by Haller in [Hal01] and it aims at extracting LCS’s from available particle paths without using the velocity field that generated the trajectories. It relies on the Finite-Time version of the Lyapunov exponent, a well known measure used to describe dynamic systems. In this section, we give its definition and report the approach proposed by Haller.

**Lyapunov Exponent: Definition and Applications**

In mathematics, the *Lyapunov exponent* (LE) of a dynamic system is a measure of the rate of separation of infinitesimally close trajectories. Two trajectories in the phase space (the space of all possible configurations of a system) which are initially at a separation $\delta S_0$ will exponentially diverge in time following

$$|\delta S(t)| = e^{\lambda t} \delta S_0$$ (1.5)
1. Introduction

where $\lambda$ is the LE. In general, for each system, there are as many LE’s as the number of dimensions of the system’s phase space and we are often interested in the maximum among them, the so called *maximum Lyapunov exponent* (MLE). A way to see the MLE, mentioned in [CAM+05], is as a measure of sensitivity to initial conditions, the larger $\lambda$, the more difficult it gets to predict the configuration of a system after a certain period of time. Applications of FE are straightforward, it gives an insight of how chaotic a system is, therefore it is used in chaos theory, where a system with a positive MLE is considered potentially chaotic. It is therefore a measure of entropy of a dynamic system.

Since simulations often do not provide an infinite time data set, some finite versions of the LE are used: the finite-time Lyapunov exponent (FTLE) and the finite-size Lyapunov exponent (FSLE). The former approximates the LE looking at particle trajectories developed in a fixed amount of time, the latter uses trajectories of adaptive length such that they separate by a prescribed factor. In next section we see how FTLE can be deduced by particle trajectories.

The Geometric Approach

Consider, a repelling structure $C(t)$ and a point $x_0 \in C(t)$. We also select a unit vector $e_{t_0}$ at $x_0$ which is not tangent to $C(t)$. We propagate $e_{t_0}$ along the trajectory $x(t; t_0, x_0)$ using the linearized flow map $\nabla \phi_{t_0}^t(x_0)$ to obtain the vector

$$e_t(x_0) = \nabla \phi_{t_0}^t(x_0)e_{t_0}$$  (1.6)

In order to locate repelling coherent structures, we now seek to maximize $|e_t(x_0)|$ over all possible choices of $e_{t_0}$. Using the operator norm

$$\|A\| = \max_x \frac{|Ax|}{|x|}$$  (1.7)

for a general matrix $A \in \mathbb{R}^3$, we can immediately maximize $|e_t(x_0)|$ over all choices of $e_{t_0}$:

$$\eta_{t_0}^t(x_0) = \max_{|e|=1} |\phi_{t_0}^t(x_0)e_{t_0}| = \|\nabla \phi_{t_0}^t(x_0)\|.$$  (1.8)

Now recall that for any matrix $A$, $\|A\|$ is what is called the spectral norm and is actually equal to the square root of the maximal eigenvalue of the positive definite symmetric matrix $A^T A$, which we denote by $\lambda_{\text{max}}(A^T A)$. We can therefore write

$$\eta_{t_0}^t(x_0) = \sqrt{\lambda_{\text{max}}(\nabla \phi_{t_0}^t(x_0))^T \nabla \phi_{t_0}^t(x_0)}.$$  (1.9)

For the sake of readability we use the Right Cauchy-Green deformation tensor defined as

$$\Delta(x_0) = (\nabla \phi_{t_0}^t(x_0))^T \nabla \phi_{t_0}^t(x_0)$$  (1.10)

which measures the square distance change due to deformation and we substitute $t - t_0$ with $T$.

The largest finite-time Lyapunov exponent $\sigma^T_{t_0}(x_0)$ associated with the trajectory $x(t; t_0, x_0)$ is defined as

$$\sigma^T_{t_0}(x_0) = \frac{1}{|T|} \ln \sqrt{\lambda_{\text{max}} \Delta(x_0)},$$  (1.11)
1.3. Finite-Time Lyapunov Exponent

Figure 1.3.: Flow separation and flow attachment: the unstable manifold (blue) attracts the fluid along the boundary and guides it into the interior of the domain whereas the stable manifold (red) guides the fluid in opposite direction.

Thus we can rewrite \( \eta_{t_0}^{T} (x_0) \) as

\[
\eta_{t_0}^{T} (x_0) = e^{\sigma_{t_0}^{T} (x_0)(T)}. \tag{1.12}
\]

Thus the local maxima of \( \eta_{t_0}^{T} (x_0) \) coincide with the local maxima of \( \sigma_{t_0}^{T} (x_0) \) and therefore, to locate Lagrangian coherent structures we will just have to find extrema (ridges) of the Lyapunov exponent field in forward or backward time.

The real MLE can not be computed with (1.11) starting from the points of a sampling grid, because of the exponentiality nature of the LE exposed in (1.5), which implies that errors in the discretization will grow exponentially. Nevertheless, if we have a grid which is fine enough, for a certain finite time \( t \) we can reasonably approximate \( \nabla \phi_{t_0}^{T} (x_0) \). Additionally our scope is that of large-scale behavior of the vector field, not its predictability, therefore we can easily adopt this method for the computation of FTLE.

Since they tend to separate fluid particles from the domain boundaries, attracting 2-dimensional coherent structures are also called separation surfaces, whereas repelling 2-dimensional coherent structures, which tend to convolve fluid particles to the domain boundaries, are called attachment surfaces, the interested reader will find in [SGH06] more precise and mathematical definitions. An example of a vector field that leads to both kinds of surfaces is shown in Figure 1.3.
1. Introduction

1.4. Height Ridges and Valley Lines

As stated in [Ebe96], height ridges are local maxima in a relaxed sense. Height ridges are basically $d$-dimensional manifolds in $n$-dimensional spaces. We present in Section 1.4.1 the definition of height ridges, which is formulated according to [Ebe96].

1.4.1. Conditions for Ridge Existence

Let $s$ be a scalar field, then height ridges are locations where $s$ has a local maximum in at least one direction.

The height ridge criterion can be formulated using the gradient and the Hessian of $s$. In a point $p$ belonging to a height ridge the eigenvectors associated to the $d$ largest eigenvalues $\lambda_i (i = 1, \ldots, d)$ of the Hessian point along the ridge, whereas the eigenvectors of the $(n - d)$ smallest eigenvalues $\lambda_j (j = d + 1, \ldots, n)$ point orthogonally to the ridge. In the point $p$ the derivatives in $\lambda_j$-eigenvector directions are zero, therefore the first condition for a ridge is that

$$\epsilon_{\lambda_j} \cdot \nabla s = 0$$  \hspace{1cm} (1.13)

with $\epsilon_{\lambda_j}$ the eigenvector associated to $\lambda_j$, i.e. there is a local extremum in the direction of this eigenvector. The second condition for a height ridge is that the second derivatives in $\epsilon_{\lambda_j}$ directions are negative, formulated as

$$\lambda_j < 0,$$  \hspace{1cm} (1.14)

i.e., neighboring points are lower values, since $\lambda_j$ is the eigenvalue of the Hessian, which express the second derivative in the direction of the respective eigenvector. Valley lines, the opposite of height ridges, are obtained by computing height ridges of the field $-s$. The reader is referred to [Ebe96], [Lin96], and [Maj00] for further details.

1.4.2. Ridge Extraction and Filtering

Since the quantity evaluated in (1.13) is scalar, one would like to extract 2D ridges with simple methods, as Marching Cubes. The problem relies in the lack of orientation of eigenvectors, evaluating such a quantity on the nodes of a sampling grid, we can incur in inconsistencies, given by differently oriented eigenvectors. The method called Marching Ridges proposed in [FP01] uses Principal Component Analysis (PCA) to achieve local consistency of the eigenvectors of a cell and it will be the chosen method for the extraction of ridges in this work.

As mentioned in Section 1.4.1, ridge extraction involves the computation of Hessian matrices and therefore of second derivatives of a scalar field. This implies a possibly large noise amplification. Smoothing is applied in these cases in order to obtain significant visualizations. As in [SP07b], smoothing is realized by incorporating it into the gradient computation, i.e., the
1.4. Height Ridges and Valley Lines

Gradient at a given node of the sampling grid is computed by fitting a linear vector field to its neighboring nodes in a Least Squares sense.

To avoid a too large number of triangles (or ridges) to be generated by the Marching Ridges algorithm, some filtering techniques are applied. These techniques are applied in a post-processing phase, only for the sake of good visualizations, they are not a central part of this work and they are part of [SP07b]. First of all, since FTLE measures the amount of separation, we can filter less relevant ridges by setting a threshold value for the ridges and visualize only those with a FTLE higher than the threshold. It is a perfectly physically motivated approach and can be easily understood by engineers, therefore it is the favorite approach for filtering. Another criterion for filtering is the dimension of the connected components of the final mesh: small components are more likely to be noise and irrelevant for visualization.
1. Introduction
Related Work

In this chapter we survey works in the scientific community that are related to this thesis. We organize it by the subjects of the works we review. These include height ridges (in Section 2.1), adaptive mesh refinement (Section 2.2), measures of mesh deformations (in Section 2.3), and importance-driven particle techniques (Section 2.4). We then briefly summarize Newtonian coherent criteria (which are opposed to the Lagrangian criterion we adopt, the finite-time Lyapunov exponent) in Section 2.5 and cite an interesting vortex definition which is related to Lagrangian hyperbolicity in Section 2.6.

2.1. Height Ridges

Height ridges are extensively treated in the literature. Our reference for the definitions given in Section 1.4.1 was the book by Eberly [Ebe96], but the works by Haralick [Har01] and [Lin96] proposed similar definitions. Other ridge concepts include Profile Ridges and Second Derivative Ridges, which are defined in the thesis by Majer [Maj00].

For the extraction of such ridges, of great importance for us is the paper by Furst et al. [FP01], in which Marching Ridges is proposed, a method which reduces the problem of representing ridges to a triangulation of an isosurface in a scalar field. In Section 1.4.2 we gave a brief overview of the algorithm and explained that it uses PCA to consistently orient eigenvectors. Another approach to achieve per-cell eigenvector consistency is proposed by Kindlmann et al. in [KTW06]. The same paper is important for the subject of height ridges, since it shows the first 2D ridges results, these were extracted from anisotropy creases in diffusion tensors of Magnetic Resonance Imaging (MRI) data.
2. Related Work

2.2. Adaptive Mesh Refinement

Adaptive Mesh Refinement was applied to the visualization of LCS’s by Sadlo et al. in [SP07a]. This approach presents a substantial speed-up of computation using a coarse grid for a first estimation of ridges and refining only those regions which satisfy certain criteria. A problem with this approach is that for the FTLE computation an evaluation of the gradient of the scalar field is needed and this is highly sampling-dependent. In other words, there is a consistent risk of underestimating the gradient on a coarse grid.

2.3. Measures of Mesh Deformation

In our method we estimate the distortion of a cell in the sampling grid by means of simple geometric measures, based on volume or edge length variation. An interesting but more expensive approach would be the one proposed by Branets et al. in [BC05]. For hexahedral (trilinear) elements, the method would derive the distortion metric from 64 basis triples, each one taking different edges as basis vectors.

2.4. Importance-Driven Particle Techniques

In [BKKW08], Bürger et al. suggested some techniques to automatically seed particles for visual representation of vector fields. Among these, the exploitation of regions with high FTLE values for the positioning of so-called anchor lines is proposed, even though credit is due to Garth et al. which exposed the idea first, in [GGTH07]. This idea is based on the assumptions that observers are interested in regions where the particles tend to diverge, and not in those where they do not diverge, where a smaller number of primitive can already give a good picture of what is happening.

2.5. Newtonian Coherent Criteria

Opposed to the Lagrangian approaches like the one we are using, there are Newtonian approaches for analysis of vector fields. We cite here the works by Hunt et al. ([HWM88]), by Chong et al. ([CPC90]), and by Jeong et al. ([JH95]), where the $Q$-criterion, the $\Delta$-criterion and the $\lambda_2$-criterion respectively, are proposed. Recall that Newtonian approaches are derived from the velocity gradient and therefore Galilean invariant, while Lagrangian approaches (based on trajectories) are objective, i.e., even invariant to accelerated rotation or translation of the frame of reference.
2.6. Objective Definition of Vortices

In [Hal04], Haller develops a criterion for defining vortices which is frame-independent. This is called the $M_Z$-criterion and identifies hyperbolic trajectories. He also shows the relation between Eulerian hyperbolicity to Lagrangian hyperbolicity, i.e., that any trajectory staying in the Eulerian hyperbolic domain is Lagrangian hyperbolic and therefore there are material lines and surfaces (see Section 1.2.1) associated to it.
2. Related Work
Method

After explaining the motivation for our approach, in this chapter we give an overview of our method and list all the important operations it consists of. We distinguish between operations that globally modify the computation grid and operations that only modify the grid locally. We describe their functionality and the criteria which trigger them during the method, along with the supporting data structures.

3.1. Motivation

Computation of trajectory-based quantities is often based on the integration of a very high number of trajectories. This reflects itself in a very high computational cost, especially for time dependent velocity fields, where for each simulation step the trajectories have to be recomputed from scratch starting from the nodes of a given sampling grid, which in classic approaches has a fixed position, i.e., the sampling grid on which we compute the quantity and hence the trajectories is always the same, for each time step. Recall from Section 1.3 that the finite-time Lyapunov exponent can be computed directly from an arbitrarily dense set of positions, in this case, then, it will always be computed on node positions.

Since we are focusing on time-dependent dynamical systems, we are interested in what we call the FTLE time series of the system. We obtain this by extracting the desired information from the system at $t_0 + k\Delta t, \ k = 0, \ldots, S - 1$, where $t_0$ is the initial time as defined in (1.11), $\Delta t$ is the temporal interval between two steps and $S$ denotes the total number of steps we want to visualize. Each element of the time series will be called computation or time step for the rest of the work.

On the one hand, our method tries to minimize computation time by restricting the grid to
3. Method

interesting regions. The input grid is grown and shrunk in order to respectively extend over ridges which are partially or totally located in its initial volume and avoid areas where there are no ridges. These structural modifications are exposed in Section 3.3.

On the other hand, our method exploits the temporal coherence of trajectory-based quantities, and tries to reduce the computation cost by reusing parts of the trajectories for more than one computation. This strategy is the core of our method and it is exposed in Section 3.4.

3.2. Overview

Our algorithm is based on a fixed sequence of operations. The operations are triggered on the basis of geometrical criteria, some of which are user parametrized. The operations -briefly described and listed in the order they are considered at each computation step- are:

- *resampling*, creation of a new sampling grid;
- *grid advection*, advection of the computational grid nodes;
- *growing*, insertion of new cells into the grid;
- *refinement*, subdivision of the grid cells;
- *pruning*, removal of cells.

This order holds for each step of the algorithm (which means for each step where we want to visualize the trajectory based quantity), notice that on the initial sampling grid growing, refinement and pruning are performed, this allows our computation to have the best possible grid as a starting point. When new nodes are added to the sampling grid, their trajectory is computed and stored in an \( N \times S + S_T \) array, where \( N \) is the total number of nodes and \( S_T \) the total number of steps we want to visualize plus the number of time steps that compose a trajectory, see Section 5.4 for a more detailed description. In the following sections, each operation is described in detail along with the criteria responsible for its triggering.

3.3. Structural Modifications

Structural modifications are all the operations which modify the overall structure of the sampling grid. This can happen by insertion (Section 3.3.2) or deletion (Section 3.3.4) of cells and nodes, by refinement (Section 3.3.3) of the sampling grid or directly by the creation of a new sampling grid (Section 3.3.1).

3.3.1. Grid Resampling

Advecting the nodes obviously deforms the cells, this does not hinder computation, since the method we use (described in [PS08]) works on generic hexahedral cells. Nevertheless cells which are too much distorted tend to add artifacts to the computation: cells can deform and
become concave, their volume can increase or decrease depending on the flow regimes they
are in and therefore the sampling can become too irregular, causing problems to, e.g., gradient
calculation, which is needed for quantities as the FTLE. For these reasons we provide the user
with control over a resampling procedure which can be triggered at any computation step in
order to sample a new grid. At the end of the resampling step, the grid has the structure which
is an arbitrarily shaped subset of what we call a reference grid. We define such a grid before
proceeding with the description of the resampling algorithm. Notice that the computational grid
at the first time step is a subset of the reference grid too.

Reference Grid

There are some operations which are heavily used by the algorithm. Such operations include
\( n \)-neighborhood search for cells (identification and location of those cells which are connected
to the \((n-1)\)-neighborhood of cell, where the 0-neighborhood is the cell itself) and similarly
for nodes and location of nodes belonging to a cell, which are repeated many times at each
computation step, as it will be shown in Section 3.3.2, Section 5.3 and Section 5.4. In order not
to lose the gain in performances of our algorithm for these kind of operations and at the same
time keeping the memory consumption as low as possible, we resorted to what we will call
reference grids. As reference grid we mean a sampling grid in which all the nodes are aligned
along the three spatial dimensions. This means that our reference grid is composed by cuboids,
even though in most application the same spatial sampling along all axes should be chosen,
turning the cells into cubes. As reference for the dimension of the cells, a cell of the initial grid
is chosen, in order to keep the consistency among reference grids at different computation steps.
Advantages of the reference grid is that each cell is uniquely identified by the three indices
\( i, j, \) and \( k \), which represent the relative position of the cell in the grid with respect to the reference
cell. In Figure 3.1, a simple 2D representation clarifies the idea of this indexing. Therefore,
for each cell, we don’t have to store the \( n \)-neighborhood but only a map which links the three
indices to the cell identifier for the unstructured grid data structure (as it will be explained in
Section 5.2.3). For a given cell \( c \) with indices \( i_c, j_c, k_c \), the neighborhood relationship \( N_n(c) \) is
then trivial. For each cell \( d \) with indices \( i_d, j_d, k_d \) it holds:

\[
d \in N_n(c) \Leftrightarrow (i_c - n \leq i_d \leq i_c + n) \land (j_c - n \leq j_d \leq j_c + n) \land (k_c - n \leq k_d \leq k_c + n) \quad (3.1)
\]
i.e., \( d \) is neighbor of \( c \) if all its indices differ from those of \( c \) at most by \( n \). This relationship is
easily proven to exhibit symmetry and reflexivity, which are the two properties which we would
expect for a neighborhood relationship.

We keep the indices of existing cells in a sorted data structure (e.g. an STL map in C++) which
links the \( ijk \) indices to the cell indices used by the main unstructured grid data structure used
to store the grid. Searching the grid for the neighbors of a given cell reduces then to searching
in the map data structure for the indices of possible neighbors. Since the map is sorted by the
indices, looking up a given index is done in \( \log I \log J \log K \) average time, where \( I, J, \) and \( K \)
are the maximum number of indices in the three directions of the grid, as shown in Figure 3.1.
Searching for the nodes of a given cell is also easy, if we index the nodes the same way we
3. Method

index the cells, taking as a reference node the first node of the reference cell. For a node \( v \) with indices \( i_v, j_v, k_v \) and a cell \( c \) it holds:

\[
v \in c \Leftrightarrow (i_c \leq i_v \leq i_c + 1) \land (j_c \leq j_v \leq j_c + 1) \land (k_c \leq k_v \leq k_c + 1),
\]

where \( v \in c \) indicates that the node \( v \) belongs to cell \( c \).

Criteria for Grid Resampling

We provide the user with two geometrical criteria for measuring grid distortion, these are:

- cell volume deviation from a reference volume;
- cell edges length deviation from a reference length.

As reference volume (or edge) the algorithm takes that of a cell belonging to the last computed reference grid. For the first criterion, the determinant of the three edges sharing one vertex and that of the three edges sharing the opposite vertex are computed. The determinant which deviates most from the reference volume is then taken as an estimator for the cell volume. The cell distortion is therefore the relative difference of the volume estimator and the reference volume. For the second criterion, the algorithm takes the cell edge which deviates the most from the reference edge and computes the relative difference of the former and the latter as the cell distortion.
3.3. Structural Modifications

Figure 3.2.: Grid resampling procedure: around the initial (dashed) and the advected (solid) grid a bounding box is created, then only the overlapping cells (and their nodes) are created. Thin dashed arrows show advection of nodes from last resampling step (at \( t = t_r \)) to current step (\( t = t_r + \Delta t \)).

Wall Cells

One important case of cell distortion happens at the domain boundaries, because a no-slip condition was imposed during the CFD simulations. Nodes very close to the boundary tend to stick to it, while the rest of the cell gets advected with the flow. We explicitly provide the user with the possibility of ignoring such cells in the measurement of the grid distortion in order to prevent their deletion (Section 3.3.4) or the resampling of the entire grid as explained in next section.

Volume Distortion in Divergence-Free Vector Fields

The CFD datasets we used for our computations are divergence-free. Mathematically speaking, this means that \( \text{div} V = 0 \), where the divergence operator \( \text{div} \) is defined as the sum of the partial derivatives, i.e. in \( \mathbb{R}^3 \) \( \text{div} V \equiv \partial V_1 / \partial x_1 + V_2 / \partial x_2 + V_3 / \partial x_3 \). A divergence-free vector field is also called volume-preserving, which means, as stated in [Asi93] that its flow \( \phi_t \) carries any open set \( S \) in its domain to a set \( \phi_t(S) \) of the same volume, for all times \( t \). It is then unclear why we observe deviations from the reference volume. We hypothesize that one cell of which two or more nodes are distant enough in a direction not tangent to the flow can undergo some kind of distortion. This could indeed be the case for cells which are close to repelling manifolds. Another source of volume deviation is obviously given by wall cells, as explained in Section 3.3.1.
3. Method

Algorithm for Resampling

The decision for resampling is based on the two cited distortion criteria, but there is an ulterior distinction, the user can decide whether to resample when the grid is globally or locally distorted. In the first case, an average of the cells distortion is computed and compared to a threshold; in the second case, the maximal distortion among all cells is compared to the threshold. In both cases, if the threshold is exceeded, the resampling is triggered.

The resampling computes the reference grid starting from the bounding box of the advected grid and the original sampling grid (the grid at the first time step). In the reference grid, only the cells which overlap with those of the advected grid or those of the initial grid are created, we show this in Figure 3.2. For each node then the trajectory is computed and stored as it will be defined in Section 5.4.

3.3.2. Growing the Grid

The growth of the grid is especially useful when extracting height ridges (see Section 1.4). The grid is grown in order to make such computation possible and to extend the computation over whole ridges, even if these were not totally inside the grid at the beginning of the computation. We will call ridge cell a cell which has at least one edge which is intersected by the filtered ridge identified according to Marching Ridges, as explained in Section 1.4.2. The set of all ridge cells will be denoted by $\mathcal{R}$.

The advantage of a regular sampling along the three axes when adding cells is that the algorithm is never in a condition of collision or overlap of cells, while with a deformed grid it would have to find an optimal size of each cell it adds and this would add a consistent overhead to the algorithm. Therefore growing of the grid is always done on the reference grid, which means that we look at the node positions at the last resample step and add the new ones (if there are new nodes) following the reference regular spacing. The new nodes are then advected to the time step where the insertion is performed. This approach has a drawback: if a certain cell is outside of the domain boundary when added to the regular grid, its nodes can not be advected and the cell can not be used in the computation. We address this problem in Section 3.3.2 and show how our approach works in this case. Another advantage of the reference grid is that it is a consistent sampling over the whole domain, this means that even starting with two regions of interest and growing the grids around them does not put us in the condition of non-matching surfaces and guarantees that the two grids can eventually merge into one. Both advantages are shown in Figure 3.3.

Criteria for Growing the Grid

Searching for height ridges on a finite sampling grid poses two problems:

- ridges extending further than the grid boundaries are truncated;
- ridges which are close to the boundaries can not be computed if they don’t have a complete neighborhood of the size needed for the gradient and Hessian computation.
3.3. Structural Modifications

We address the two issues together. The algorithm identifies all the ridge cells and tries to surround them with a complete neighborhood.

**Algorithm for Growing the Grid**

Once the algorithm has identified all the ridge cells which have an incomplete neighborhood (according to the range we need for our computation), it looks for the missing cells, simply looking up each hypothetical neighbor cell in the map described Section 3.3.1 and marking those which do not exist. For each cell that virtually has to be added, the node indices are computed (as described in Section 3.3.1). For each node we have the following possibilities:

- the node belongs to the sampling grid and it is inside the domain boundaries;
- the node belongs to the sampling grid, but it is outside of the domain boundaries;
- the node does not belong to the sampling grid.

In the first case no operation has to be taken, if all nodes of a cell exist, then the cell is simply built with them as its vertices. In the second case the algorithm is trying to add a cell which at the initial position (or at the last resampled position, see Section 3.3.1) is outside of the boundaries with at least one of its vertices, this is a special case we address in next section. In the third case, if the node is inside the domain boundaries it is inserted into the grid, its trajectory is computed and stored, then the algorithm proceeds to the next node. If the node the algorithm is trying to add is outside of the boundaries, we are again in the case addressed in next section.

The growing is iterated as long as for each ridge cell, wherever it is possible, the neighborhood needed by the computation is fulfilled. This means that each newly added cell is tested for the presence of a ridge and, if it contains a ridge, the algorithm tries to complete its neighborhood.

**Growing toward Walls**

When the algorithm tries to add a cell which has a node outside of the domain (recall that we are doing such an operation on the reference grid as explained in Section 3.3.1), it can try to move the node to a position inside the domain boundaries. To do this, the algorithm looks for the neighbor which has the trajectory which best approximates the edge connecting it to the node that has to be moved inside the boundary. This is done by testing the velocity field at each neighboring node position. We prefer testing the velocity field rather than the trajectory ending at that point, because trajectories could be curved and, if the velocity is high with respect to the time step, we could have to subsample the trajectory and get a less accurate estimate of the flow direction. The neighbor where the velocity field best approximates the edge is chosen as a candidate for the move. If the velocity field and the hypothetical edge form an angle of less than $30^\circ$ (which was empirically chosen to limit the possible divergence from orthogonal cells and meanwhile guarantee an amount of freedom to the algorithm), then the algorithm evaluates the time which would take the node to describe a trajectory of the length of the edge. The algorithm then goes along the trajectory of the candidate node for that time and stores the position as the initial position of the node which has been recovered. In Figure 3.4 an example of such a procedure is depicted.
Figure 3.3.: Growing the Grid: inserting cells in advected grids poses two problems. In blue, position of new nodes can not be extrapolated from its neighbors; in red, the nodes of a cell connecting two separated grids (both originating from the same reference grid) are not trivial to identify. Both problems are solved looking at the reference grid, positions for new nodes are straight forward, they are aligned to the reference grid, and cells connecting two separated regions are simply those which connect them in the reference configuration. The final result is obtained by advection of newly inserted nodes.
3.3. Structural Modifications

Figure 3.4.: Growing Toward Walls. (a) After advection of the sampling grid, gaps between the boundaries and the grid itself may appear. (b) Moving the problem to the reference grid does not help, since the nodes needed for growing are outside in this configuration. (c) and (d) Trajectories of nodes which would be neighbors of the new ones are exploited, in order to create new cells for the advected grid. (e) We now have to update the reference grid. We follow the trajectories of the newly inserted nodes until they reach the positions of the reference grid. (f) Newly inserted nodes are advected and the reference grid is grown.
3. **Method**

The advantage of such a procedure is that we are always sure that the algorithm will find a consistent position for the node (the trajectory can not go outside of the domain boundaries, it is truncated on the boundary itself). The drawbacks are that the reference grid will also have non-cuboids cells and that most of the cells which will be inserted this way will be wall cells and will get highly distorted during advection (see Section 3.3.1 for a detailed description of this phenomenon).

### 3.3.3. Grid Refinement

For each ridge cell we allow the user to refine the grid, in order to have a higher level of detail. Each cell is split into eight cells, generated by the planes containing the midpoint of the cell edges. The refinement is performed on the reference grid (see Section 3.3.1), this has two advantages.

The first advantage of refining on the reference grid is that we gain consistency. Since we are computing on hexahedral cells, planar surfaces are not guaranteed. This means that interpolating the four nodes belonging to one advected face will almost never give points which lie on the face itself. Therefore building the four subdivided faces will lead to big differences from the non subdivided one. Interpolating on the reference grid will, on the contrary, create perfectly subdivided faces, since faces of cuboids are indeed planar. The consequent advection of the nodes will then produce subdivided faces which are more similar to (though, as we explain in next section, still different from) the non-subdivided one. The second advantage is that refinement on the reference grid produces less distorted cells, compared to the original ones. This can be easily explained if we think of the gradient of the vector field $\nabla u$, if two nodes are distant, in a continuously differentiable vector field (which indeed is our case), their velocity can be much more different than two nodes which are closer. Obviously different velocities of nodes tend to distort the cell, therefore, while geometrically subdividing on the advected cell would (also in the case of planar cells) just create shrunk versions of the original cell, subdividing on reference grid and then advecting the nodes will lead to (most likely) changes in shape of the cells and less distortion.

Even though the second cited advantage seems to contradict the first one, let us point out that the combination of the two leads to smaller variation of non planar surfaces and better shapes (in the sense of littler amount of distortion) of deformed cells. The two advantages are depicted in Figure 3.5.

It must be noticed that after a refinement step occurred, the reference grid is changed accordingly to the new dimensions of the reference cell. This means that all the successive reference grids will have cells with halved edges with respect to those of time steps preceding the refinement step.

### 3.3.4. Pruning the Grid

Shrinking the grid is important for the reduction of computation time. For example, cells which do not belong to the $n$-neighborhood of a ridge cell, denoted by $\mathcal{N}_n(\mathcal{R})$, are deleted.
3.3. Structural Modifications

Figure 3.5.: Subdivision. (a) In case of 3D non planar cells, interpolating the vertices leads to incorrect subdivision and sampling of the space. (b) Interpolating on the reference grid makes possible for nodes to follow the flow and create cells which are better approximations of initial ones. (c) Subdividing on edge midpoints creates cells which look like shrunk versions of the initial one, with no attention paid to the different flows they are in. (d) Subdividing on reference grid creates nodes which are then advected with the local flow and can generate less distorted cells.

Criteria for Pruning the Grid

At each time step the algorithm identifies cells which are to be removed from the grid. This can happen depending on a set of the following criteria:

- one or more nodes of the cells were advected outside of the domain boundaries;
- the cell is not a ridge cell and does not belong to the \( n \)-neighborhood of such a cell;
- the cell became concave due to advection of its nodes;
- the distortion of a cell due to the advection of its nodes is above a certain threshold.

Each decision criterion is independent from the others, it is easy therefore to parametrize the choice of the algorithm according to the analysis requirements of the investigator.

Algorithm for Pruning the Grid

For the first three criteria cited above, a cell is always deleted, for the fourth one a distinction is made by the algorithm depending on the user’s choice. The user can in fact decide whether to delete distorted cells or keep them inside the grid. The same thing holds for wall cells (see Section 3.3.1), if the user excludes them from distortion measurement, they will never be deleted due to this criterion. An example of the pruning procedure is shown in Figure 3.6.
3. Method

Figure 3.6.: Pruning the Grid. An example of the pruning procedure, the area outside of the domain boundaries is shadowed in gray. The chosen criteria for pruning are those imposing to delete cells which are not in the 1-neighborhood of ridge cells (crossed with orange dashed lines) and those of which at least one node was advected outside the boundary (crossed with red dashed lines). 1-neighborhood was chosen for illustration reasons.

The deletion is then trivial, cells are just removed from the sampling grid, but nothing hinders that a deleted cell will be re-inserted into the grid at a later simulation step, since this could indeed be the case for an unsteady vector field.

3.3.5. Region of Interest

Since it is not always easy to foresee a good region of the domain for ridge extraction (because it is difficult to estimate the Lyapunov exponent of a system a priori), or because only part of the ridge is of interest, one could want to restrict the growing to a specified Region of Interest (ROI) and set slightly weaker bounds to ridge extraction and consequent growing of the computational grid. The grid will then be grown more easily following ridges but will be limited by the ROI boundaries. This means that cells outside of the ROI boundaries will not have a complete $n$-neighborhood, but they will still be added even if they are outside the ROI boundaries, if they are needed to complete the neighborhood of a cell inside the ROI.

3.4. Grid Advection

Recall the setting exposed in (1.2), we have an initial condition $x(t_0) = x_0$ at time $t_0$. If at each step we keep the same node positions, we can not exploit the consistency condition, because at the time $t$ generally no node will have a position which is the end of the trajectory starting at another node position time $= t_0$. Since we are interested in reusing the old trajectories, to reduce the computations, we advect the nodes of the sampling grid with the flow. This way at each time we will be able to use the consistency condition to our advantage. Consider the case where we start from $t = t_0$ and compute the FTLE on trajectories which extend on a
3.4. Grid Advection

Figure 3.7.: Grid Advection. At time step \( t_1 \), the part of the trajectories computed for each node starting from time step \( t_{n-t} \) (where \( n \) here denotes the number of steps needed for the computation of a trajectory) and finishing at \( t_0 \) is reused in order to compute the trajectory starting at time step \( t_{n+1} \) and finishing at the same time step \( t_0 \). To obtain the whole trajectory, only the part going from \( t_0 \) to \( t_1 \) has to be computed. The trajectory-based feature (in this case, an FTLE height ridge, constituting a separation surface) can then be extracted.

time interval of \( T \), with discrete integration steps \( \Delta t \). Then, with the definition of the flow transformation \( \phi \) given in Section 1.1.1, for a node satisfying the initial condition it holds \( x(t_0 + T + \Delta t) = \phi_{t_0 + T}(x_0) \) and for the following integration step, \( x(t_0 + T) = \phi_{t_0 + T + \Delta t} \circ \phi_{t_0 + T}(x_0) \).

In general we know from the consistency condition that

\[
\phi_{t_0 + T + \Delta t} = \phi_{t_0 + T + \Delta t} \circ \phi_{t_0 + T}, \tag{3.3}
\]

but this can also be decomposed as follows:

\[
\phi_{t_0 + T + \Delta t} = \phi_{t_0 + T + \Delta t} \circ \phi_{t_0 + T} \circ \phi_{t_0 + \Delta t} = \phi_{t_0 + T + \Delta t} \circ \phi_{t_0 + \Delta t} \circ \phi_{t_0 + T \circ \phi_{t_0 + \Delta t}}. \tag{3.4}
\]

The first equivalence, intuitively said, means that following the trajectory of a point for a time \( T \) and then for a time \( \Delta t \) is the same thing as following it for \( \Delta t \) and then for \( T \). The second equivalence means that we can follow the trajectory for \( \Delta t \), then for the time we need to arrive at time \( T \) (this interval will then be of a time length \( T - \Delta t \)) and then for again \( \Delta t \). Thus, if we store the trajectory of any node from \( t_0 \) to \( T \), we just need to compute \( \phi_{t_0 + T + \Delta t} \), i.e. integrate for \( \Delta t \), and add this path to the already computed trajectory starting at \( t_0 + \Delta t \) and finishing at \( T \). A schematic representation of this concept is shown in Figure 3.7, while practical results of advected grids can be seen in Chapter 4.
3. Method

3.5. Information Storage and Reload

Our algorithm is animation-oriented, this means that we are looking for an output which can be easily used to look at different time steps with one run of the algorithm. To do this with a classic approach, one would have to store at each computation step the information to be shown. Depending on the desired output, one can store the complete trajectory for each node or the derived quantity only. A sample execution of the algorithm with both phases, computation of 6 simulation steps and display of a given step, is depicted in Figure 3.8.

Stored Information

Our algorithm stores the complete trajectories of the nodes at each time step. Since our grid is dynamic and changing at each time step, information about its structure has to be stored for future retrieval. To do this, we rely on the sequentiality of our operations and the consistency of nodes and cells indices when re-applying the same operations to the same initial state. Therefore, the information we store is essentially the structure of each reference grid we build (the initial sampling and each resampled grid) and, for each step, the cells we add, the number of refinement steps we perform and the cells we delete. We then keep a list of all the steps where a resampling took place.

Retrieved Information

After the computation steps, to retrieve data of a given time step we look at the last resampling step before it, load the reference grid and then sequentially perform the operations which took place from the resampling step until the time step we want to display. Since the trajectories are stored, we can then extract the height ridges and represent them graphically.
3.5. Information Storage and Reload

Figure 3.8.: A sample execution of the complete algorithm.
3. Method
Results

In this chapter we present the application and the results of our method on different unsteady CFD simulations which were provided by partners of the ETHZ. Section 4.1 presents three examples, each one introducing a different case of study, while Section 4.2 describes some analytical aspects of the algorithm, along with some more examples.

4.1. Method Application

In this chapter we present applications of our method to three different CFD simulated data sets. The data sets investigated in Section 4.1.1 and Section 4.1.3 are courtesy of VA Tech Hydro AG, while the data for Section 4.1.2 is courtesy of Sulzer Innotec. All the simulations where obtained with ANSYS CFX.

4.1.1. Von Kármán Vortex Street off a Runner Blade

A von Kármán Vortex street is a periodically repeating pattern of swirling vortices caused by separation of flow over round shaped objects. It is well known in engineering, because due to its behavior it can apply periodical forces to the object originating it and, if these forces have a period which is close to the object’s resonance frequency, undesirable effects, such as cracks, can arise.

In water turbines von Kármán vortex shedding may occur at the trailing edge of stay vanes, guide vanes, and even at the trailing edge of runner blades. In order to analyze the stay vane failure in an existing power plant in Canada, CFD simulations of von Kármán vortex shedding
were carried out for the geometry and flow conditions of the power plant. Additional investigations were done in order to analyze the effect of different trailing edge geometries on the intensity of vortex shedding \cite{LKS+06}.

Here we analyze one of these simulations using FTLE ridges. Garth et al. \cite{GLTH07} have already demonstrated the utility of FTLE analysis for a vortex street. Because high spatial resolution is necessary for capturing the phenomenon in this case, only the flow around a small section of the blade is analyzed, see Figure 4.1 and Figure 4.2. Some performance results are shown in Table 4.1. The achieved speed-up is quite low (3.73) because short trajectories were used compared to the length of the advection step.

![Von Kármán vortex street. (a) Geometry. (b) Sampling grid after adaptation and advection.](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Direct</th>
<th>Grid Advection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid [nodes]</td>
<td>9699</td>
<td>9699 (at step 15)</td>
</tr>
<tr>
<td>Flow map [s]</td>
<td>5562.69</td>
<td>959.02</td>
</tr>
<tr>
<td>Total [s]</td>
<td>5563.32</td>
<td>1488.45</td>
</tr>
<tr>
<td>Speed-up factor</td>
<td>1</td>
<td>3.73</td>
</tr>
</tbody>
</table>

Table 4.1.: Performance analysis for the Kármán dataset. 20 steps of grid advection compared to 21 direct evaluations on advected grid. See also Figure 4.1(b).

### 4.1.2. Flow around a Cuboid

This example produces a vortex shed which is similar to a von Kármán vortex street of the type shown in Section 4.1.1. The flow comes from the right back and follows to the left front.
4.1. Method Application

Figure 4.2.: Von Kármán vortex street. (a) Resulting FTLE ridge. (b) Resulting FTLE ridge and some upstream trajectories (colored) from a uniform grid with seeds (white spheres).

(Figure 4.3(a)). The real difference compared to the example of Section 4.1.1 is that there exists a “top” face of the cuboid, whereas the runner blade spanned the vertical domain. The result is a very different flow separation behavior compared to the simulation of Section 4.1.1. A von Kármán vortex street can be recognized as separating from the cuboid, but the vortex axes tilt and tend to get horizontal. An isosurface of the $\lambda_2$ [JH95] was used for visualizing the vortex street (Figure 4.3(b)) because, differently from many other vortex detection methods, it works fine with transversally moving vortices.

The resulting FTLE ridge (Figure 4.4(b)) shows that flow separation happens on both sides and on the top of the cuboid. It can be seen that the FTLE ridge separates the vortex street region (attached to the face of the cuboid) from the outer flow, but this property disappears further downstream and the FTLE ridge crosses the vortices. Table 4.2 shows some performance details for this example. The achieved speed-up in this case is about 5. Because of the shape of the FTLE ridge, the relatively low sampling resolution (Figure 4.4(a)), and because the initial sampling grid is already well adapted to the FTLE ridge, the expected speed-up due to grid adaptation is small and was therefore not measured.

4.1.3. Intake of a Power Plant

We analyze in this section a data set representing an existing run-of-river plant in the US. The problem there is that entering of juvenile salmons in the intake should be prevented. For this reasons, some structures exist. These same structures originate separation surfaces in the flow. The water flow of the unsteady CFD simulation comes from the right back and follows to the left front where it enters the turbine (Figure 4.5(a)).
4. Results

Figure 4.3.: Flow around a cuboid. (a) Geometry. (b) Isosurface of $\lambda_2$ at level $-15000$, showing flow separation and vortex street.

The horizontal rods at the right hand side of the image lead the salmons into the vertical channels at the top in the installation. However, these rods produce a noticeable wake in the upper part of the main channel (see path lines in Figure 4.5(b)). Additionally, the backflow from the salmon channel (the opening at the top downstream from the rods) also is involved in a recirculation zone at the top wall, located above the sampling grid of Figure 4.5(b). On the one hand, a FTLE ridge was extracted using a regular grid at the confluence of the three main channels (Figure 4.5(b)), on the other it was extracted using the presented grid advection method (Figure 4.6(a) and 4.6(b)). The obtained FTLE ridge well separates the fast flow at the bottom of the channel from the slower flow in the upper half of the channel.

Table 4.3 shows some performance measurements of the presented case. The speed-up due to grid adaptation is quite low (about a factor of 2) because of the relatively low resolution of the sampling grid and because the sampling region of the uniform grid is already quite well adapted to the ridge. The speed-up due to grid advection is significantly higher (11.79) and would further increase with increasing the integration time for the trajectories.

<table>
<thead>
<tr>
<th></th>
<th>uniform</th>
<th>grid advection</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid [nodes]</td>
<td>3718</td>
<td>2610 (at step 7)</td>
</tr>
<tr>
<td>flow map [s]</td>
<td>610.68</td>
<td>66.57</td>
</tr>
<tr>
<td>total [s]</td>
<td>611.1</td>
<td>123.46</td>
</tr>
<tr>
<td>speed-up factor</td>
<td>1</td>
<td>4.94</td>
</tr>
</tbody>
</table>

Table 4.2.: Performance analysis for the cuboid dataset. 20 steps of grid advection compared to 21 direct evaluations on uniform grid. See also Figure 4.4(a).
4.2. Performance Analysis

Figure 4.4.: Flow around a cuboid. (a) Sampling grid adapted to ridge region and advected. (b) Resulting FTLE ridge with some upstream trajectories (colored) from uniform grid, and their seeds (white spheres).

<table>
<thead>
<tr>
<th></th>
<th>uniform</th>
<th>direct on adapted grid</th>
<th>grid advection</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid [nodes]</td>
<td>8800</td>
<td>5034</td>
<td>5034 (at step 3)</td>
</tr>
<tr>
<td>flow map [s]</td>
<td>3166.8</td>
<td>1604.4</td>
<td>205.07</td>
</tr>
<tr>
<td>total [s]</td>
<td>3167.85</td>
<td>1604.61</td>
<td>268.62</td>
</tr>
<tr>
<td>speed-up factor</td>
<td>1</td>
<td>1.97</td>
<td>11.79</td>
</tr>
<tr>
<td>Figure</td>
<td>4.5(b)</td>
<td>4.6(a)</td>
<td>4.6(a)</td>
</tr>
</tbody>
</table>

Table 4.3.: Performance analysis for the turbine intake dataset. 20 steps of grid advection compared to 21 direct evaluations (on uniform grid and adapted grid). See also Figure 4.5(b) and Figure 4.6(a).

4.2. Performance Analysis

Our method is based on precise integration of particle trajectories in large vector fields. This is the dominating part of the algorithm and it does not make sense to do a deep asymptotic analysis of it, since it depends on too many factors and it (obviously) scales almost linearly in the number of particles (i.e. grid nodes) and steps. Therefore we looked for an improvement of basic, highly repeated, time consuming operations, such as neighborhood search and point location. This was done partially with ad hoc data structures and partially by algorithmic ideas, such as the reference grid.

We ran some simulations aimed at determining possible bottlenecks, results are that the most clear source of time overhead is the resampling procedure. In fact, each step in which a resampling occurs is exactly the same as the computations of classic algorithm and does not exploit time coherence. For example, looking at Figure 4.7 it is clear how each resampling step is slow.
4. Results

Figure 4.5.: Intake of a water turbine (clipped). (a) Geometry. (b) Uniform grid with some of the upstream trajectories (colored) used for FTLE computation, and their seeds (white spheres).

The analyzed set is the same of Section 4.1.2, the number of nodes and cells at each time step is shown in Figure 4.8, we computed FTLE using trajectories of 50 time steps and extracted it in an interval of 40 steps. For a normal computation step we measured an average time of 58.3 seconds, whereas a resampling step took 498.1 seconds, which means a factor of about 8.5 for the two different kinds of steps, even though less nodes are advected.

Then, why do resampling at all? One reason for resampling is that cells which are too much distorted tend to create artifacts in the resulting ridges, this can be seen looking at the time series of the FTLE ridges, from the step preceding a resampling, to the step where this occurs, we observe so-called “popping artifacts”, inconsistencies between temporarily subsequent surfaces occur. We discuss this issue in Section 6.2. Popping artifacts are the reason why we provided the user with measures of grid distortion. What we discovered is that even in divergence-free flows cell volumes tend to variate. As it can be seen in Figure 4.9, the average value increases linearly at each time step, which is to a certain degree strange, since we would have expected it to grow exponentially. We also analyzed the data set of Section 4.1.3, and we get the same kind of linear growing of average volume distortion, as it is shown in Figure 4.10.
4.2. Performance Analysis

Figure 4.6.: Intake of a water turbine: grid advection. (a) Sampling grid adapted to ridge region and advected. (b) Resulting FTLE ridge.

Figure 4.7.: Computation time: resampling steps are easily identified by the slope of the curve, which is more than eight times higher that of normal advection steps.
4. Results

\[\text{Figure 4.8.: Number of nodes and cells: the total number of nodes and cells at each computation step of our algorithm. Notice that resampling shrinks both number of nodes and of cells.}\]
4.2. Performance Analysis

Figure 4.9.: Volume distortion for cuboid data set: it can be seen that average distortion grows linearly in time. For the measurement also wall cells where considered.
Figure 4.10.: Volume distortion for intake data set: another example of the linear growing of the average distortion. In this case, wall cells where not considered for measuring distortion.
Algorithm Implementation

In this chapter we describe implementation issues of our algorithm. It is thought as a guide for further development or for a re-implementation of the method.

5.1. The Framework

We implemented our method as an FTLE ridge extraction module in Advanced Visualization System (AVS), a data visualization software for engineering and business applications. It is based on modules networks where data flows from module to module through user-defined connections. We describe the usage and the ports of our module in Chapter A. The module is implemented in C++.

5.2. Sampling Grid

Our module relies on a sampling grids. We therefore defined some supporting data structures to make access and manipulation of such grids easier.

5.2.1. Unstructured Cell Data Library

As it will be explained in Section A, the input and output sampling grids of our algorithm are of type UCD_structure, which is the AVS native object type for unstructured grids. It is defined in the Unstructured Cell Data (UCD) library and it is often used for results from Finite
5. Algorithm Implementation

Element Methods (FEM) and CFD simulations. We refer the interested reader to the AVS Developer’s Guide [AVS92] for details on the class and will enumerate only features which will be relevant for our work.

In an object of type UCD_structure, positions of grid nodes are stored in interleaved arrays, the same thing holds for data at the nodes. For each cell a connectivity list is stored, containing the indices of the nodes of a cell and, for each node, it is possible to store the cells sharing it. Nodes always have indices which go from 0 to N-1, where N is the total number of nodes, “holes” (indices pointing to non-existing nodes) are not allowed and a similar reasoning holds for cells. Supported cell types are triangles, quadrilaterals, lines, tetrahedra, pyramids, prisms, hexahedra and points. We only used hexahedra for our sampling grids.

The UCD_structure library provides the user with basic routines for querying and modifying cells and nodes (positions and data are stored in interleaved arrays, their access and manipulation is allowed, though not always intuitive), but suffers of a complete lack of topological routines. As mentioned in Section 3.3.1, for our algorithm neighborhood searches and relationships are of extreme importance, and in the library there is no routine for querying such information. Another important missing feature is one about point location, given a point, there is no straight forward way to know whether it is inside a grid cell or not. Moreover, since we add and remove cells continuously and “holes” are not allowed in the indexing, the same cell can have much different indices in two subsequent computation steps and these changes of indices can become very hard and expensive to track. For these reasons we had to create auxiliary data structures for our computation.

5.2.2. Unstructured Grid Wrapper

The Unstructured class is a wrapper for objects of type UCD_structure, it provides recursive neighborhood searches and a point location routine based on spatial optimized search. Although these could be enough for our computation, we experienced long time overheads due to the recursive neighborhood search, which has to find relationships between nodes and cells, since these are not stored anywhere. Furthermore, even though this wrapper solves two problems, the third one, the one about indices inconsistencies from one time step to another is not solved. We therefore used this wrapper for point location, and recurred to a STL map to circumvent the other two problems.

5.2.3. Mapping Indices to Cells and Nodes

As mentioned in Section 3.3.1 we keep one mapping between three indices and the relative cell, and one between indices and relative node. This was done using the STL map and implementing a small class indices which makes possible to sort every triple according to a lexicographical order. The two maps are then defined as std::map<indices, int> and they are updated at each iteration of the algorithm in an easy way, since this is done on a reference grid: for each node the index in any direction is the number of rows (or columns, or slabs) separating it from a chosen reference node common for all nodes. For the cells, we proceed similarly, using as references the cell centroids.
5.3. Neighborhood

With this data structures we can then easily find neighbors disregarding changes in the indices and following the method defined in Section 3.3.1.

5.3. Neighborhood

During the growing phase described in Section 3.3.2, to do our neighborhood search in a convenient way we build the complete neighborhood relationship for all the cells. This means that for an \( n \)-neighborhood we build an \( H \times (2n+1) \times (2n+1) \times (2n+1) \) array, where \( H \) is the number of hexahedral cells, and \( (2n+1) \times (2n+1) \times (2n+1) \) is the size of one cell’s complete neighborhood. We take advantage of symmetry of the relationship and if for a cell \( c \) we find \( d \) as a neighboring cell, then we update \( d \)'s neighborhood in order not to look up the position of cell \( c \) while looking for \( d \)'s neighbors (remember that we are interested in those positions in the neighborhood which are free and there must be inserted a cell).

5.4. Trajectories, Node Positions and Resampling

The implemented module outputs trajectories for the grid nodes, these can be useful and/or requested in flow visualization, e.g for enriching the FTLE visualization. We will explain in Section A.2.8 how this is parametrized.

We illustrate an example of a hypothetical run of the algorithm to show how the node positions are stored and how track of the different configurations is kept. Let us first consider a simple situation where no shrinking, growing or refinement is triggered. The example is depicted in Figure 5.1. Then, let \( C \) be the current time step during a computation and \( R \) be the step where the last resampling occurred (could be 0, if we are using the initial sampling grid as reference). Let further \( B \) be the buffer where we store the trajectories of each node (the so called buffer array). Then \( B \) has to be an \( N \times (S-R+S_T) \) array, where \( S \) is the total number of advection steps (see Section A.2.14 for details of its parametrization), \( S_T \) is the number of steps we use for the trajectories computation (see Section A.2.10), \( N \) the total number of nodes. This way, if no other resampling occurs, \( B \) has exactly the space needed for storing node positions needed for trajectory computation, grid advection and future grid retrieval. If at the computation step \( R' \) another resampling occurs, then we need to store a new array \( B' \), because the number of nodes will in general be different from one resampling step to another. Then, \( B \) was obviously over-sized, because the number of steps it has to contain is no more \( S-R+S_T \), but \( R'-R+S_T \). We therefore copy \( B \) to a right sized array and create \( B' \) as an \( N' \times (S-R'+S_T) \), where \( N' \) denotes the number of nodes in the newly resampled grid. The last step is then to keep a vector \( V \) of pointers to the different buffers, which will then be used to set the grid positions in the retrieval phase.

When the grid is grown or refined, then \( B \) gets reallocated with a larger capacity, in order to adapt to the new total number of nodes. When nodes are deleted, then \( B \) is not affected, since for those nodes we still have to keep trajectories preceding their deletion. Trajectories will then be created as objects of type \texttt{AVSfield_float} and filled with positions taken from the right buffer, at the desired time steps. To be able to reconstruct reference grids in the retrieval phase,
Figure 5.1.: Trajectories storage example from Section 5.4. (a) At the step $R$ a resampling occurs, gray shaded elements of buffer $B$ are position stored only for trajectories computation, they are not node positions of the advected grid at any computation step, pale blue elements are node positions of the advected grid at each step of the time series we are computing. $V$ is the vector containing pointers to previous buffers and the relative resampling step. (b) At the step $R'$ another resampling occurs, the buffer $B$ is shrunk to the necessary size, while a new buffer $B'$ is created. A pointer to $B$ is stored in $V$, along with $R$, which was the step where the $B$ was created. (c) Computation ends at step $S$. A pointer to $B'$ is stored in $V$, along with $R'$. 
5.5. Insertions, Refinements and Deletions

For each resampling step we have to store their configuration: we store $I$, $J$, and $K$ as they are described in Section 3.3.1 and a list of the cells not to keep (since most of the times this list will be smaller than the one of cells to keep).

5.5. Insertions, Refinements and Deletions

For each modification we perform during a computation step, we have to store information about changes in the grid structure. For an insertion we store information for the new cell such as the number of nodes which have to be inserted into the grid in order to build such a cell, for the refinement of one cell we just have to store the cell index, since then refinement can be executed on the reference grid yielding the same result in the retrieval phase as in the computation, for deletion, again, only the deleted cell index has to be saved. We keep track of the executed operations in three distinct vectors (one for each type of modification). Notice that the operations are saved sequentially. In addition to this, for each operation and at each time step, we save the index in the vector of the first operation we perform. A little example should clarify the idea: let the computation be at a certain step $C$ and the three vectors $I$, $R$, and $D$ storing insertions, refinements and deletions information respectively. Let further $i = 4$, $r = 5$, and $d = 2$ be the length of the vectors $I$, $R$, and $D$ respectively. Let assume vectors to be 0-indexed, as it is in the C++ standard. Then we keep three arrays of pointers $P_I$, $P_R$, and $P_D$, where $P_k$ is the array of pointers related to vector $k$. If we perform all three operations, then we will store, at the position $C - 1$ in the arrays, 4, 5, and 2 respectively. During the retrieval phase (see Section 3.5), if we will have to execute the operations of this step we will jump to the right vectors elements for all the operations and execute all the operations belonging to this step (to do this, the computation step where each operation was performed is stored, in order to stop executing operations when they belong to another step).
5. Algorithm Implementation
Conclusion and Outlook

This chapter is a final discussion about our work. In Section 6.1 we expose the achieved results compared to what we had as target for this work, in Section 6.2 we suggest possible improvements and in Section 6.3 we illustrate future direction for research in this field.

6.1. Initial Goals and Achieved Results

We discuss in this section the expectations we had on this work and how they were met by the results.

We aimed at improving the computation time achieved by existing methods and this was a success. Even though we would have probably thought of a higher speed-up factor, we demonstrated that this approach works and is valuable. Things we could not foresee were those which implied consistent bottlenecks for our computation, like e.g. resampling. These are obviously the motivation for the initial overestimation of speed-up factor. Meanwhile, we knew we had to find good strategies for issues like matching of different (disjoint) advected grids and formation of gaps at boundaries, we found solutions to these problems, by using the reference grid and growing toward walls, therefore we consider this target as accomplished.

6.2. Possible Improvements

As we already stated, one of the big bottlenecks of our algorithm is the resampling procedure. We described drawbacks of it in Section 4.2 and we explained that grid distortion tends to create artifacts. This could sound strange, since theoretically we are extracting a mathematically and
physically meaningful quantity and we are interpolating it linearly on the cell. What we think is that probably the way we represent ridges is not optimal.

We recall that the marching ridges algorithm works as follows: for each node the quantity $\epsilon_{\lambda_j} \cdot \nabla s$ defined as in Section 1.4.2 after consistently orienting the eigenvectors, is computed and then isosurfaces of this new scalar quantity are computed, interpolating the values along the edges of the grid. Now the problem could arise from the fact that we are interpolating linearly a quantity that, in fact, is not linear, since it is derived by a dot product of a gradient and an eigenvector of a Hessian matrix. We therefore suggest to use a different approach, which is based on the bisection method. We still look for the point where (1.13) holds, but we do this by computing a component-wise linear interpolation of the gradient and the Hessian matrix at the midpoint of the edge and then recursively subdividing the edge, taking the half edge which has different signs of $\epsilon_{\lambda_j} \cdot \nabla s$ at its defining vertices, where $\epsilon_{\lambda_j}$ and $\nabla s$ denote the eigenvector of the interpolated Hessian and the interpolated gradient respectively.

### 6.3. Future Work

Since the results are quite encouraging, this work will be continued. Future work will include the implementation of our approach for ridge extraction, new improvements of the supporting data structures and strategies for locally reducing the distortion of the grid, in order to have less frequent resampling steps.
AVS User Interface

As mentioned in Section 5.1, we implemented our method as an AVS module for FTLE ridge extraction. In this chapter we explain the User Interface (UI) of the module, i.e., the input and output ports of the module itself and the usage of the parameters visible in the control panel. Figure A.1 shows the module and for each port the section containing the relative explanation is indicated. Figure A.2 shows the control panel, for each parameter widget the section containing the relative explanation is indicated.

The reference figure for this section is Figure A.1.

Figure A.1.: AVS module: for each input and output port the respective section is indicated.
A. AVS User Interface

A.1. Input and Output Ports

In this section we show the connection ports of the implemented module, define the types of input and output along with their use.

A.1.1. Velocity Field

This port takes as input an AVS UCD_structure object which must contain the CFD velocity field from which the FTLE has to be extracted.

A.1.2. Initial Sampling Grid

This port takes as input an AVS UCD_structure object which must be the initial user defined sampling grid. The grid must be built following a convention: the nodes of each cell must be in AVS order and the first node of each cell must be the one with the smallest value in $x$ and $y$ direction, and the greatest value in $z$ direction. Further, the grid must nodes must be aligned as in a reference grid (see Section 3.3.1 for further explanations).

A.1.3. Distance from Walls

This port takes as input an AVS UCD_structure object which must contain information about the distance of each point of the domain from the no-slip walls.

A.1.4. Distortion

Through this port the module outputs the grid at the selected (see Section A.2.15) animation step, in form of an AVS UCD_structure object, containing per-node information about the average distortion (Section A.2.18) of the cells that the node belongs to.

A.1.5. Trajectories

Through this port the module outputs the trajectories of each node at the selected animation step of the output sampling grid, in form of an AVS AVSfield_float object. The output depends on the state of the option complete trajectories (see Section A.2.8).

A.1.6. Output Grid

Through this port the module outputs the grid at the selected (see Section A.2.15) animation step, in form of an AVS UCD_structure object, containing per-node information about FTLE, and for debugging and additional insight, its eigenvalues, flow map (see Section 1.3),
and integration time of the trajectory starting at that node. To avoid retriggering of downstream dependent modules, the output of this port is set to “unchanged” (a built-in feature of AVS) when the grid is not modified by changes of the module parameters.

A.2. Control Panel

The control panel combines the module’s widgets and is the interface where the user can parametrize the algorithm execution. The reference figure for this section is Figure A.2.

A.2.1. Velocity File Selector

In this widget the component of the UCD from Section A.1.1 containing the vector field for steady data has to be selected.

A.2.2. Unsteady Toggle

This button toggles the computation of the LCS’s between steady (time-independent) and unsteady (time dependent) vector fields.

A.2.3. Unsteady Velocity File Selector

In this widget the file containing information about the unsteady vector field has to be loaded. This widget only appears if the option unsteady is enabled (see Section A.2.2).

A.2.4. maximum Level of Subdivision

This parameter is a positive integer and it sets the level of subdivision that is to be reached for each cell.

A.2.5. Start Time

This parameter is a floating point number, indicating the initial time where the computation will start in the temporal domain of an unsteady vector field (it corresponds to $t_0$ for the FTLE computation of Section 1.3). It must belong to the temporal domain covered by the file loaded in the unsteady velocity file selector described in Section A.2.3.
Figure A.2.: AVS control panel: for each parameter the respective section is indicated.
A.2.6. Region of Interest Scale

We provide control over the grid expansion by means of a simple floating point scale factor which has to be given as this parameter and determines the maximum growing in each direction with respect to the initial computation grid: a scale factor of 0 will limit the grid to the original extent (enlarged of the number of cells needed to complete the neighborhood of each cell); a scale factor of 0.5 will allow a growing of the grid of half of the original extension in each direction (thus each grid dimension will at most be doubled and the maximum grid size will be eight times the original size). This is a simple implementation of the region of interest described in Section 3.3.5.

A.2.7. Forward Toggle

This button toggles the direction for the integration of trajectories, i.e., if it is enabled, then trajectories will be integrated forward in time and the grid will be advected backwards, whereas if it is disabled trajectories will be integrated backward in time and the grid will be advected forwards (see Section 1.2.1 and Section 1.3 for an explanation of the effects of the different integration directions).

A.2.8. Complete Trajectories Toggle

This button toggles the display of complete trajectories, it determines the output of the trajectories port (see Section 5.4). If the option is disabled, the represented trajectories are those extending (where possible, if the particles were not advected outside of the domain) on the time interval used for computation of the FTLE, if it is enabled, the represented trajectories are the complete integrated trajectories, possibly extending further than the time requested for computation of the FTLE.

A.2.9. Integration Time T

This parameter is a floating point number which determines the time over which a single trajectories extends. It is $T$ defined in Section 3.4.

A.2.10. Integration Steps

This parameter is an integer number which determines in how many time steps the algorithm will subdivide each trajectory for representation purposes (this includes the time interval between two subsequent computations in the time series). Notice that the $\Delta t$ defined in Section 3.4 can be defined as the parameter given in Section A.2.9 divided by the number given as this parameter. Also notice that $\Delta t$ is not the time step for integration, because this is done using adaptive step size.
A. AVS User Interface

A.2.11. Minimum Scalar for Ridge Extraction

This parameter is a floating point number determining the lower threshold value for FTLE ridge extraction, used for the detection of ridge cells, as they are defined in Section 3.3.2. It is the same threshold defined in Section 1.4.2.

A.2.12. Maximum Scalar for Ridge Extraction

Similar to Section A.2.11, this parameter is a floating point number determining the upper threshold value for FTLE ridge extraction.

A.2.13. Smoothing Range for Ridge Extraction

This parameter is an integer number determining the range of nodes on which the smoothing described in Section 1.4.2 is applied. It also indirectly determines the \( n \) dimension of the \( n \)-neighborhood defined in Section 3.3.1, in fact, \( n \) is equal to the value of this parameter plus one, because the gradient neighborhood for FTLE computation is 1.


This parameter is an integer number determining the number of computation steps which will be computed by the algorithm. It also determines the time interval over which the grid is advected, i.e., the time interval over which the FTLE ridges are extracted, together with the integration time \( T \) and the number of integration steps, defined in Section A.2.9 and in Section A.2.10 respectively: in fact the total computation time is \( \Delta t \) (as defined in Section A.2.10) times the number of computation steps, i.e., the value of this parameter.

A.2.15. Animation Step

This parameter is an integer number determining the computation step to retrieve information of. It can take values between 0 and the value of advection steps and its function is depicted in the right part of Figure 3.8.

A.2.16. Ignore Wall Cells Toggle

This button toggles the distortion measurement of wall cells (see Section 3.3.1), when active, wall cells are ignored during distortion evaluation.
A.2.17. Distortion Tolerance for Resampling

This parameter is a floating point number determining the tolerance threshold for grid resampling as defined in Section 3.3.1.

A.2.18. Distortion Criterion Radio Buttons

This buttons represents the possible criteria for sampling grid distortion measure as defined in Section 3.3.1. *volume* and *volume max* set the criterion to global and local volume distortion respectively, whereas *length* and *length max* set the criterion to global and local edge length distortion respectively.

A.2.19. Delete Distorted Cells Toggle

This button toggles the deletion of distorted cells in the pruning procedure, as described in Section 3.3.4. If active, distorted cells are deleted, if inactive, they are kept in the grid.

A.2.20. Distortion Tolerance for Deletion

This parameter is a floating point number determining the tolerance threshold for cell deletion as defined in Section 3.3.4. If *delete distorted cells* is active, as described in Section A.2.19, then cells exceeding this distortion threshold are deleted from the grid.

A.2.21. Growing toward Walls Toggle

This button toggles the growing of the grid toward walls, as described in Section 3.3.2.
A. AVS User Interface
Bibliography


Bibliography

Springer, accepted for publication, 2007.


Introduction

Lagrangian methods in numerical flow visualization, such as the Finite-Time Lyapunov Exponent (FTLE), often require the integration of a very high number of trajectories. Usually one trajectory needs to be computed for each node of the computational grid. This constitutes a very high cost especially for transient vector fields because there the trajectories need to get recomputed for each time step. This thesis addresses the problem by exploiting temporal coherency by reusing part of the trajectories. This can be achieved by advecting the nodes of the computational grid. The main goal of this thesis is to develop a method for the efficient visualization of unsteady separation phenomena in CFD simulations, based on FTLE.

Task / Work Packages

• Advection of computational grids and FTLE computation thereof
• Development of methods to handle the advection at the (no-slip) boundaries
• Development and application of metrics for advection error and cell degeneracy
• Possibly application of the method to other Lagrangian quantities such as Mz

Requirements

• C/C++ programming skills
• Sufficient mathematical skills

Remarks

A written report and an oral presentation conclude the work. The thesis is overseen by Prof. Markus Gross and supervised by Filip Sadlo, Institute of Computational Science. For further information or application to this project, please contact Filip Sadlo, IFW C27.1, Tel. 632 71 44, sadlo@inf.ethz.ch.

Time-Frame

Hand in: 9. April 2008